Internet Security Analysis and Audit

CSE3501

Slot – L25+L26

Final Project Review

Title:

SQL Injection Detection and Prevention System

Members:

Hiya Kulasrestha - 20BCE2828

Francin Samuel - 20BCE2836

Zamaan Saleel - 20BCE2025

Abstract

The goal of our research is to design and create an internet buying management system that is secure from various attacks. Since the majority of used goods are classified and internet purchasing is growing in popularity, both the buyer and the seller must come to an agreement about risks regarding fraud. We are therefore, including detection and prevention techniques to safeguard the website and all its users from external SQLI or XSS attacks. Users should find it safe and convenient to buy goods online with the online shopping management system. The main purpose of this project is to make products which are useful for students available for purchase. It is implemented in order for students to save money and help them in financial terms. We are also implementing speech recognition for this website. The ability of a machine or program to recognise words spoken aloud and translate them into legible text is known as voice recognition, often known as speech-to-text. The concept of a hands-free future in which a computer's mouse and keyboard are superfluous has been introduced by speech-to-text technology. Keywords: , SQL Injection, XSS Attacks, Speech Recognition

Introduction

Web applications and online platforms for purchasing goods and services have become an ever-growing phenomenon. This has also brought with it the rising risk of security risks, some of them being SQL Injection attacks and Cross-Site Scripting (XSS) attacks. SQL injection is an attack in which the hacker subverts application logic by changing a query. This can be used to access or modify confidential data from the web application's private database. On the other hand, XSS attacks are a form of attack in which the hacker injects malicious scripts via the medium of a trusted website, which compromises the interactions of that website with other end users. It is imperative that any web application that is created is efficiently protected against these kinds of attacks in order to preserve website integrity and user confidentiality and trust. Hence, we must implement appropriate countermeasures, making generous use of suitable detection and prevention techniques and algorithms. Several such measures are already in use in many applications, yet with higher complexity of risks, we must also improve the efficiency of our algorithms to ensure maximum security.

Literature Survey

S. No.	Title	Technique	Advantages	Limitations
1.	A survey of SQL injection attack detection and prevention, Khaled Elshazly1 Et al., Journal of Computer and Communications (2014) [1]	User data sensitive keyword filtering, Apache server's rewrite module, LDAP protocol, PHP Data Object and Prepared Statements, SQLMAP vulnerability scanner for penetration testing	● Any flaws in the configuration of database privileges or coding of the application would not affect the database security ● Real time analysis does not impact the database	● Pilferage of data ● Not suitable with large number of data set ● Significant false-negative rate
2.	Securing e-commerce against SQL injection, cross site scripting and broken authentication, Ng Yi Xuan Et al., Journal of Applied Technology and Innovation (2021) [2]	SQL, SQLIA, SQL Parser, SQL Check Attacker, JDBC-Checker, CANDID, SQL Guard and SQL Check, Swaddler, Positive tainting, WebSSARI, AMNESIA	● Security features are also functioning well in preventing the designed payloads for each attack ● Secures against SQLi, XSS and broken authentication with simple steps taken	● Can bypass without knowing the right password because it is always true ● Unauthorized access database ● Fails to transform some particular code
3.	Protection of E-Commerce Website from SQL Injection: A Review, Meena, International Journal of Scientific & Engineering Research (2016) [3]	SQL, Java script, PHP, CSS, SQLI, Authentication,Man in the middle attack, Content Security Policy, Input Sanitization, Secure Authentication Configuration, Entities Encoding, Hashing and Salting	● can detect all types of attacks at both static and dynamic phase	● It is more towards information handling ● It is not handling user inputs ● Account Hijacking
4.	SQL Injection Attacks Prevention System Technology: Review, Fairoz Q Kareem, AJRCS (2021) [4]	SQL, Database, Authentication, DDL, DML, Secure Sphere, Mod Security, Airlock, Airlock, Green SQL, Dot Defender, Rational Unified Process (RUP) with the aid of Unified Mark-up Language (UML).	● the Web application systems developed in other languages can also use the practices described in this article for security testing	● Protecting dynamic input are not trivial ● Passively detecting SQL injection is not as useful for preventing in real time applications ● Use of packet sniffer does not allow for the SQL injection prevention from malicious code.
5.	SQL Injection Attack Detection and Prevention Techniques Using Machine Learning, Ines Jemal, IJAER (2020) [5]	Forming SQL injection string pattern, designation of parse tree, detection of SQL injection and XSS attacks using Knuth-Morris-Pratt algorithm, formulation of filter functions	● Is lightweight and can fit a large number of requests per second ● Ensures the semantic relations between the different parts of the attack signature	● Fails to prevent encoded injection ● Algorithms demand more data than classic techniques

6.	A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm, Oluwakemi Christiana Abikoye, EURASIP Journal on Information Security (2020) [6]	Query-model and obfuscation based SQLI countermeasures, CANDID tool, SEPTIC approach, Random4 algorithm, AutoRand implementation	● can successfully detect and prevent the attacks, log the attack entry in the database, block the system using its Mac Address to prevent further attack, and issue a blocked message ● not limited to a particular form of attack, and it can handle different forms of SQL injection and XSS attacks	● Cannot deal with external function calls ● Fails to transform some particular code ● Limits the user input possible values
7.	Analysis of SQL injection detection techniques, Jai Puneet Singh, CIISE (2016) [7]	In this paper they have compared various types of SQl injection detection techniques	● simplified and better protection and detection techniques while keeping in mind about the comparison of performance of different techniques related to Blink SQL Injection, Fast Flux SQL Injection and Compounded SQL Injection	● Less prevention and detection techniques ● Research needs to be done more ● Developer should have good knowledge otherwise it leads to destroy of organization
8.	Detection of SQL injection attacks using Hidden Markov Model, Debabrata Kar, IEEE International Conference on Engineering and Technology (2016) [8]	The technique used to detect SQL injection attacks is hidden markov model	● works at the database firewall layer and therefore protects multiple web applications hosted on a shared server ● computational overhead was minimized ● minimal performance impact which is imperceptible over the Internet	● Minimal performance ● Needs more data ● Needs to improve performance
9.	A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection, Rahul Johari Et al., IJATER (2012) [9]	SQL, Input Validation, Database Mapping, Static Analysis, Dynamic Analysis,HTML, CWE framework, ASPROX, Java, JavaScript, ActiveX, Flash, Hybrid Encryption.AES, RSA, Intrusion Detection System (IDS), Java Cryptography Architecture, Hashing algorithm for encryption, AMNeSIA, Obfuscation-based Analysis, Structured Query Language	● algorithm correctly replaced 94% of the SQL injection vulnerabilities in these projects ● requires no change in the runtime system, and imposes a low execution overhead	● Developer availability and learning is required ● Source code adjustment is needed. ● If the original developer left the project it is very difficult to patch the vulnerabilities.

10.	SQL Injection Prevention System, Kupershtein L.M., Et al., International Conference "Radio Electronics & InfoCommunications (2016) [10]	Validation checker, a bunch of methods to filter all input data from users that generate request signatures, Secure Shell filters the output information and blocks injection to the database, and its own error handler.	● provides a budget, automated solution for protecting Web resources on SQL databases	● Fails to transform some particular code ● Algorithms demand more data than classic techniques
11.	Semantic Query-Featured Ensemble Learning Model for SQL-Injection Attack Detection in IoT-Ecosystems, Gowtham M, Et al., IEEE Transactions on Reliability (2022) [11]	applies state-of art count-vectorizing, case-conversion and stopping word removal as preprocessing steps to ensure its suitability under an unknown analytics environment, applies different Word2Vec semantic features to train a machine learning model to detect malicious queries or SQLIA in IoT-database, performs resampling using random sampling and up-sampling algorithms, distinctly	● improved learning-efficiency and hence overall classification accuracy ● reduces the computational cost, but also improves time efficiency to meet real-time SQLIA detection demands	● its efficacy remained limited to SQL query only ● did not find the use of up-sampling and random sampling inevitable
12.	Network security education: SQL injection attacks, Srdjan Zivanic, Et al., IEEE Zooming Innovation in Consumer Technologies Conference (2022) [12]	Frontend applications have been written using HTML and CSS languages, while connection to a database is written in PHP, uses the MariaDB, fork of the MySQL database management system, Mann Whitney significant predictor test and PCA	● Does not tackle attacks with single quotes and back quotes	● Incomplete implementations ● Cannot deal with external function calls ● Minimal performance
13.	SQL Injection Detection and Prevention using Aho-Corasick Pattern Matching Algorithm, Shreya Kini, Et al., 3rd International Conference for Emerging Technology (2022) [13]	implemented using XAMPP server, a sample of well-known attack patterns is used to evaluate the proposed approach, every time an attempt to injection via input fields occur, the proposed technique is successfully able to detect and prevent the attack	● effective against Boolean based, Error based, Time based and Union based attacks ● highly implausible to generate false positives and false negatives, increasing the reliability of the approach	● randomization technique can change the intended user query and it also limits the possible user input ● Subpar results in terms of throughput and accuracy
14.	Authentication Enhancement Against SQL Injection Attacks (SQLIAs), Samar M. Albalawi, Awad M. Awadelkarim Mohamed [14]	discover entries that contain gaps.using hashing values of user,input validation, input sterilization, advanced encryption standard, collaborative authentication, input filtering, cryptographically hashed at runtime with recognition based graphical login credentials, problem detection, critical review and building solution	● develop a policy that detects SQLIA attacks that affect the authentication process,flexible solution to enhance security against SQL injection attacks to avoid authentication bypass and to reduce risk ● Its advantage is that it can block SQL injection completely or partially.	● It is mainly unreliable because it does not recognize traditional SQL injection methods. ● Where there is no clear distinction between regular query statements and query statements from SQL injection which will lead to a large number of Errors in recognition

15.	Detection of SQL Injection Attack Using Adaptive Deep Forest, M.S. Roobini, S.R. Srividhya, Sugnaya, Kannekanti Vennela, Guntumadugu Nikhila, International Conference on Communication, Computing and Internet of Things (2022) [15]	adaboost profound timberland-based calculation, SQL infusion strategy- routine examination, dynamic investigation, and separating.	● resolves the issue of expanding and debasing the first condition of the woodland. ● Our model can change the size of the tree quickly and take care of numerous issues to stay away from issues.	● There is still opportunity to get better in the WAPS-CIVS counteraction framework. ● This doesn't resolve the issue of combining XQuery with XQuery Unity. ● The need to interface counterfeit records called CSRF or XSRF is notable, however it is like a Cross Site (XSS) attack.
16.	An Sql Injection Detection Model Using Chi-Square with Classification Techniques, Marion Olubunmi Adebiyi, Micheal Olaolu Arowolo, Goodnews Ime Archibong, Moses Damilola Mshelia, International Conference on Electrical, Computer and Energy Technologies (2021) [16]	dataset, Chi-square Feature Selection and classification using Decision Tree, K Nearest Neighbor, Naive Bayes and evaluated using performance metrics(sensitivity, specificity, precision, accuracy)	● provides an unbiased evaluation of a final model fit on the training dataset. ● Selects the relevant features that will provide an improved performance, before being passed into Decision Tree, Naive Bayes and K Nearest Neighbor classifiers	● huge amounts of work make the selection of the best solution that fits a given web application a difficult task. ● It should include widening the scope of usage of the model. ● Other algorithms can be introduced such as Neural Network, Kmeans, Logistic Regression in order to improve the performance of the system.
17.	SQL Injection Detection Technology Based on BiLSTM-ATTENTION, Pengcheng Wen, Chengwan He, Wei Xiong, Jihui Liu, 4th International Conference on Robotics, Control and Automation Engineering (2021) [17]	Application of neural network method to identify the injection, using depth sequence model with loss layer, multi string analysis and word2vec, BiLSTM model.	● Accuracy of 99.3% ● recall of 98.2% ● Compared to traditional methods this method can identify SQL injection attacks more efficiently. ● Retains past and forward information	● Due to complex and variable form, it collects more than other types of SQL data to improve generalization.
18.	A GAN-based Method for Generating SQL Injection Attack Samples, Dongzhe Lu, Et al., IEEE ITAIC (2022) [18]	a sample generation method based on a genetic algorithm and deep convolutional generative adversarial network, certain random interference is introduced to the generated samples to simulate the complexity of real production environment and the diversity of attack samples to improve the performance of deep learning intrusion detection methods	● could also be widely applied to other vulnerability attack samples, such as XSS scripts and command injection scripts	● False positives and false negatives ● there are infinitely many such expressions, and it is meaningless to list them, so we have to eliminate these cases when cleaning the data

19.	The Detection and Defense Mechanism for SQL Injection Attack Based on Web Application, Li Min, Et al., Joint International Information Technology and Artificial Intelligence Conference (2022) [19]	proposes a real-time detection mechanism of SQL injection attack based on triangle module operator. In order to ensure the information system from SQL injection attacks, security testing should be carried out, including source vulnerability testing, penetration testing, etc	● we can discover the hidden dangers in time and avoid the occurrence of security incident	● Intensive manual work requirements ● the input data should be strictly controlled to avoid security loopholes
20.	SQL Injection Attack Detection by Machine Learning Classifier, Prince Roy, Et al., International Conference on Applied Artificial Intelligence and Computing (2022) [20]	Authors have used Logistic Regression, AdaBoost, Naive Bayes, XGBoost, and Random Forest to find the SQL injection payload in the dataset. To identify the best classifier for detecting SQL injection, Kaggle SQL injection data set is used.	● The proposed solution does not require complex operations like Parse trees or specific libraries	● More optimization methods, such as deep learning, can also be used. ● Complex frameworks

The analysis of our literature review reveals that the majority of existing techniques suffer from one or more of the following limitations:

- Runtime overheads
- Inherent limitations
- Complex frameworks
- Incomplete implementations
- False positives and false negatives
- Intensive manual work requirements

Existing Methods

SQL injection is an ongoing issue that was discovered a long time back. However, in the last few years, SQLIA has been on the rise. Several methods have been discovered and introduced in order to tackle this issue. Such applications have still proven to be vulnerable. There are a few applications that have been developed by companies in an effort to provide a solution to this problem. Some have been outlined below:

- JDBC-Checker: It can be used to prevent attacks that take advantage of type mismatches in a dynamically-generated query string. As most of the SQLIAs consist of syntactically and type correct queries so this technique would not catch more general forms of these attacks.

- CANDID: It modifies web applications written in Java through a program transformation. This tool dynamically mines the programmer-intended query structure on any input and detects attacks by comparing it against the structure of the actual query issued. CANDID's natural and simple approach turns out to be very powerful for detection of SQL injection attacks.

- SQL Guard and SQL Check: In these, queries are checked at runtime based on a model which is expressed as a grammar that only accepts legal queries. SQL Guard examines the structure of the query before and after the addition of user-input based on the model. In SQL Check, the model is specified independently by the developer. Both approaches use a secret key to delimit user input during parsing by the runtime checker, so security of the approach is dependent on attackers not being able to discover the key. In two approaches developers should modify code to use a special intermediate library or manually insert special markers into the code where user input is added to a dynamically generated query.

- AMNESIA: It combines static analysis and runtime monitoring. In the static phase, it builds models of the different types of queries which an application can legally generate at each point of access to the database. Queries are intercepted before they are sent to the database and are checked against the statically built models, in dynamic phase. Queries that violate the model are prevented from accessing the database. The primary limitation of this tool is that its success is dependent on the accuracy of its static analysis for building query models.

- WebSSARI: It uses static analysis to check taint flows against preconditions for sensitive functions. It works based on sanitized input that has passed through a predefined set of filters. The limitation of approach is adequate preconditions for sensitive functions cannot be accurately expressed so some filters may be omitted.

- SecuriFly: It is another tool that was implemented for java. Despite other tools, it chases string instead of character for taint information. SecurityFly tries to sanitize query strings that have been generated using tainted input but unfortunately injection in numeric fields cannot stop by this approach. Difficulty of identifying all sources of user input is the main limitation of this approach.

- Positive Tainting: It not only focuses on positive tainting rather than negative tainting but also it is automated and does need developer intervention. Moreover this approach benefits from syntax-aware evaluation, which gives developers a mechanism to regulate the usage of string data based not only on its source, but also on its syntactical role in a query string.

- IDS: It uses an Intrusion Detection System (IDS) to detect SQLIAs, based on a machine learning technique. The technique builds models of the typical queries and then at runtime, queries that do not match the model would be identified as attack. This tool detects attacks successfully but it depends on training seriously. Else, many false positives and false negatives would be generated.
  Another approach in this category is SQL-IDS which focuses on writing specifications for the web application that describe the intended structure of SQL statements that are produced by the application, and in automatically monitoring the execution of these SQL statements for violations with respect to these specifications.

- Swaddler: It analyzes the internal state of a web application. It works based on both single and multiple variables and shows an impressive way against complex attacks on web applications. First the approach describes the normal values for the application's state variables in critical points of the application's components. Then, during the detection phase, it monitors the application's execution to identify abnormal states.

- Secure Sphere: Uses advanced anomaly detection, event correlation, and a broad set of signature dictionaries to protect web applications and databases. It also uses error responses from the same user to identify an attack.

- Green SQL: Is a free Open Source database firewall that sits between the web server and the database server and is used to protect databases from SQL injection attacks. The logic is based on evaluation of SQL commands using a risk scoring matrix as well as blocking known database administrative commands (e.g., DROP, CREATE, etc.). Reports are generated on timestamp, query pattern, reason block (e.g., true expression, has "or" token). It has a white list of approved SQL patterns. However, only MySQL database is currently supported. In comparison, the IDPS in this project may be used with any relational database, not just MySQL. The IDPS has both black and white list pattern features.

Proposed Methodology

The patterns for various forms of SQL injection and XSS attacks were studied and solutions were devised based on these patterns. The methodology implemented in this study has five phases: formation of SQL injection string pattern, designing parse tree for the various forms of attacks, detecting SQL injection and XSS attacks, preventing SQL injection and XSS attacks using KMP algorithm and formulating the filter functions.

1. **Detecting SQL injection and XSS attacks**

   Detected types of SQL injection and XSS attacks:

   (i) Boolean-based SQL-injection attacks.

   (ii) Union-based SQL injection attacks.

   (iii) Error-based SQL injection attacks.

   (iv) Batch query SQL injection attacks.

   (v) Like based SQL injection attack

   (vi) XSS attack

2. **Prevention of SQL injection and XSS attacks using KMP algorithm.**

   KMP string matching algorithm was used to compare user's input string with different SQL injection and XSS attacks patterns that have been formulated.

   The Knuth-Morris-Pratt matching algorithm is one that uses a degenerating property (i.e., a pattern having same sub-patterns appearing more than once in the pattern) of the pattern and then improves the worst-case complexity to O(n). The basic idea behind this algorithm is that whenever one detects a mismatch, we already know some of the characters in the text of the next window from the previously made matches.

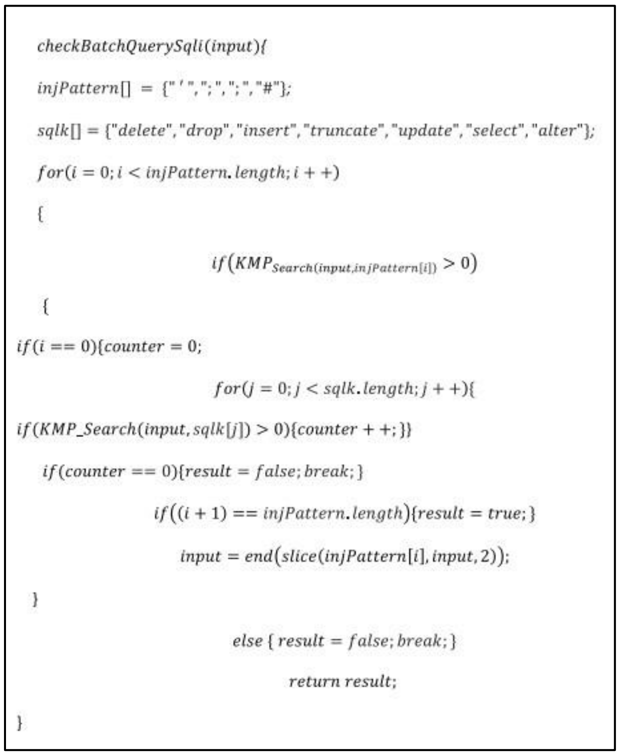
ALGORITHM:

```
$I = \sum_{i=0}^{n} f_i$    Where f is the user's input from each form text field

filter(I) { data = convertASCIItoString(I);
        if(data <> " "){
                a = checkBooleanBasedSqli(data)
                b = checkUnionBasedSqli(data);
                c = checkErrorBasedSqli(data);
                d = checkBatchQuerySqli(data);
                e = checkLikeBasedSqli(data);
                f = checkXss(data);
                    if(true (a||b||c||d||e||f)){
                        blockUser( );
                        resetHTTP( );
                        warningMessage( );
                    }
                        else  {grantAccess(); }
        }
```

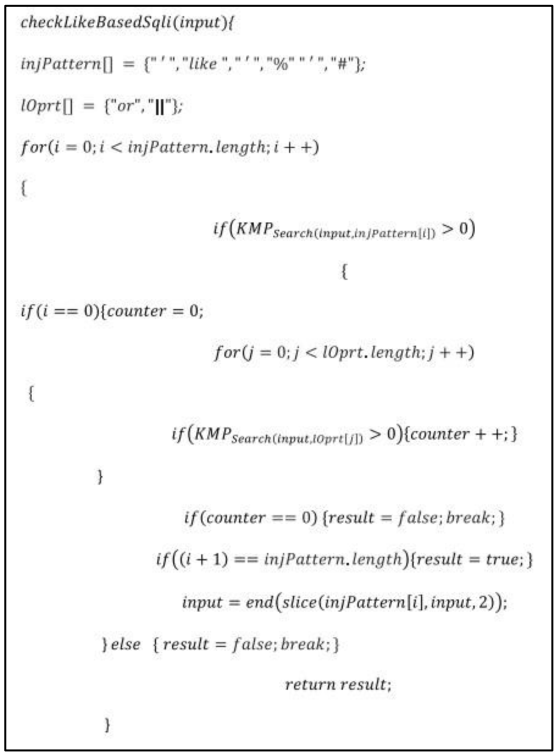
```
checkBooleanBasedSqli(input){
injPattern[] = {"'","'","'"," = ""'","'","#"};
lOprt[] = {"or","||"};
rOprt[] = {'=','>','>=','<','<=','<>','!='};
for(i = 0; i < injPattern.length; i++)
{
                        if(KMP_{Search(input,injPattern[i])} > 0)
                                {
if(i == 0){counter = 0;
                        for(j = 0; j < lOprt.length; j++){
if(KMP_Search(input, lOprt[j]) > 0){counter++;}}
        if(counter == 0){result = false; break;}
                                if(i == 2)
        {counter = 0;
                        for(k = 0; k < rOprt.length; k++){
                    if(KMP_{Search(input,rOprt[k])} > 0){counter++;}
        }
        if(counter == 0){result = false; break;}
                if((i + 1) == injPattern.length){result = true;}
                    input = end(slice(injPattern[i], input, 2));
      } else{ result = false; break;}
                                return result;
}
```

```
CheckUnionBasedSqli(input){
injPattern[] = {"'","union "," select ","from" ,"#"};
for(i = 0; i < injPattern.length; i++)
{
                        if(KMP_{Search(input,injPattern[i])} > 0)
    {
                if((i + 1) == injPattern.length){result = true;}
      input = end(slice(injPattern[i], input, 2));
    }
                                else{ result = false; break;
    }return result;
    }
```

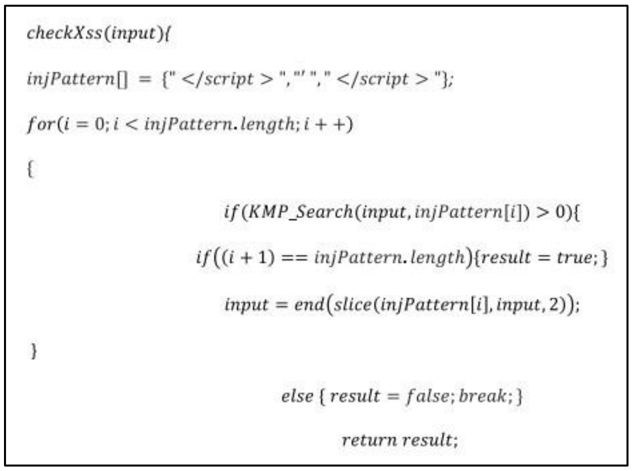
```
checkErrorBasedSqli(input){
injPattern[] = {"'",")"};
sqlFn[] = {"convert(","avg(","round(","sum(","max(","min("};
for(i = 0; i < injPattern.length; i++){
                        if(KMP_{Search(input,injPattern[i])} > 0)
                                {
if(i == 0){counter = 0;
                        for(j = 0; j < sqlFn.length; j++){
if(KMP_Search(input, sqlFn[j]) > 0){counter++;}}
                    if(counter == 0){result = false; break;}
                    if((i + 1) == injPattern.length){result = true;}
                        input = end(slice(injPattern[i], input, 2));
  }  else{ result = false; break;}
      return result;
}
```



```
    checkBatchQuerySqli(input){
    injPattern[] = {"'",";",";","#"};
    sqlk[] = {"delete","drop","insert","truncate","update","select","alter"};
    for(i = 0; i < injPattern.length; i + +)
    {
                        if(KMP_Search(input,injPattern[i]) > 0)
     {
if(i == 0){counter = 0;
                        for(j = 0; j < sqlk.length; j + +){
if(KMP_Search(input, sqlk[j]) > 0){counter + +; }}
     if(counter == 0){result = false; break; }
                  if((i + 1) == injPattern.length){result = true; }
                    input = end(slice(injPattern[i], input, 2));
   }
                           else { result = false; break; }
                                return result;
}
```



```
checkLikeBasedSqli(input){
injPattern[] = {"'","like ","'","%" "'","#"};
lOprt[] = {"or","||"};
for(i = 0; i < injPattern.length; i + +)
{
                    if(KMP_Search(input,injPattern[i]) > 0)
                                {
if(i == 0){counter = 0;
                    for(j = 0; j < lOprt.length; j + +)
 {
                 if(KMP_Search(input,lOprt[j]) > 0){counter + +; }
        }
                   if(counter == 0) {result = false; break; }
                 if((i + 1) == injPattern.length){result = true; }
                   input = end(slice(injPattern[i], input, 2));
        } else  { result = false; break; }
                              return result;
        }
```



```
  checkXss(input){
  injPattern[] = {" </script > ","'"," </script > "};
  for(i = 0; i < injPattern.length; i + +)
  {
                       if(KMP_Search(input, injPattern[i]) > 0){
                     if((i + 1) == injPattern.length){result = true; }
                       input = end(slice(injPattern[i], input, 2));
   }
                            else { result = false; break; }
                                 return result;
```

3. Formulating the filter functions

The filter() function was formulated to prevent SQL-injections and XSS attacks. This function is a combination of other functions that have been written with the purpose of at least detecting a particular form of attack. This way even if one of the functions return as "TRUE" then the filter() will block the user, reset the HTTP request, and display a corresponding warning message.

PSEUDO-CODE:

(i) Formulating the checkBooleanBasedSqli( ) function: this was used to prevent Boolean-based SQL injection attack

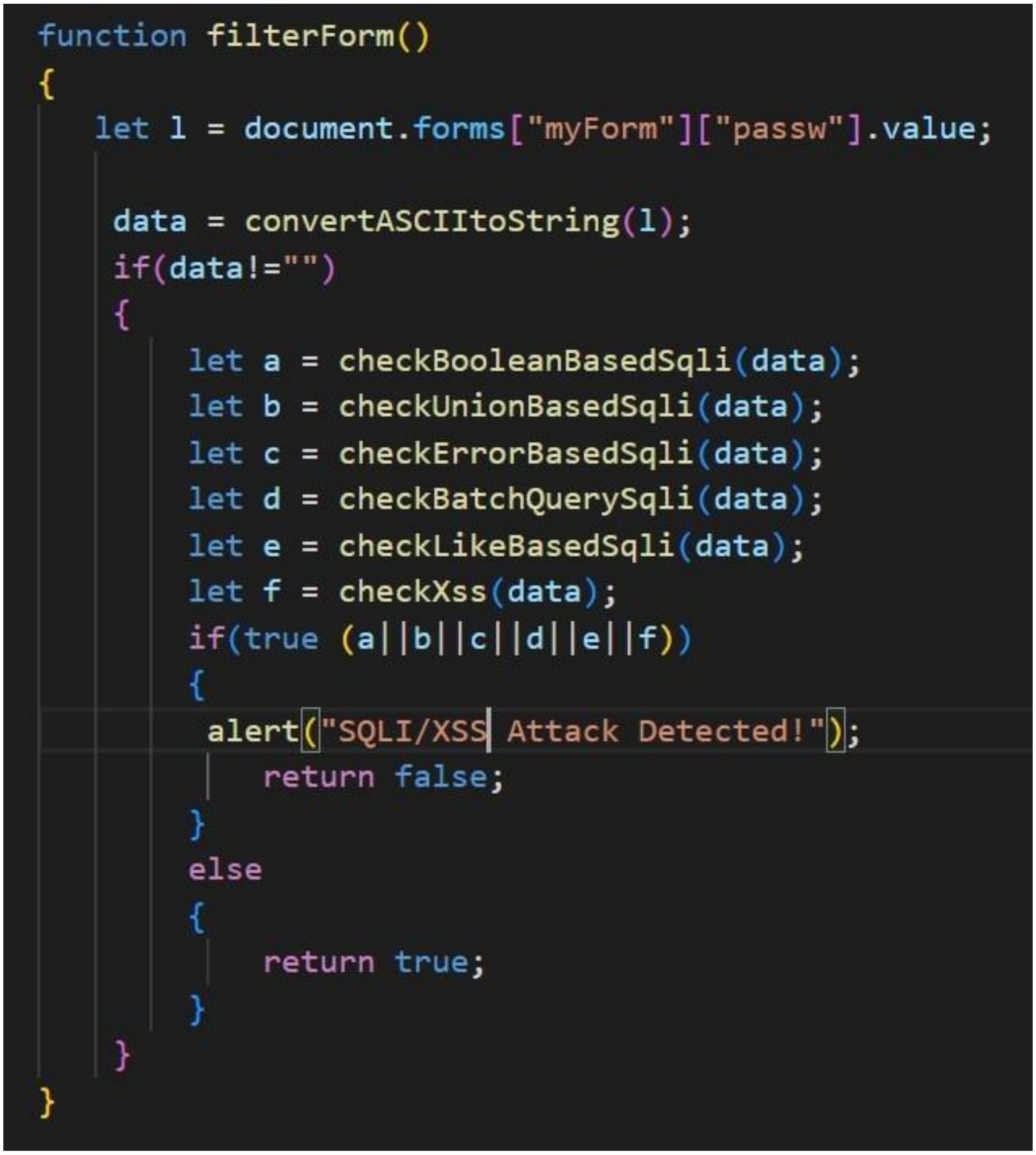
(ii) Formulating the checkUnionBasedSqli() function: this was used to prevent union-based SQL injection attack

(iii) Formulating the checkBatchQuerySqli() function: this was used to prevent batch query SQL injection attack.

(iv) Formulating the checkLikeBasedSqlis() function: this was used to prevent like-based SQL injection attack.

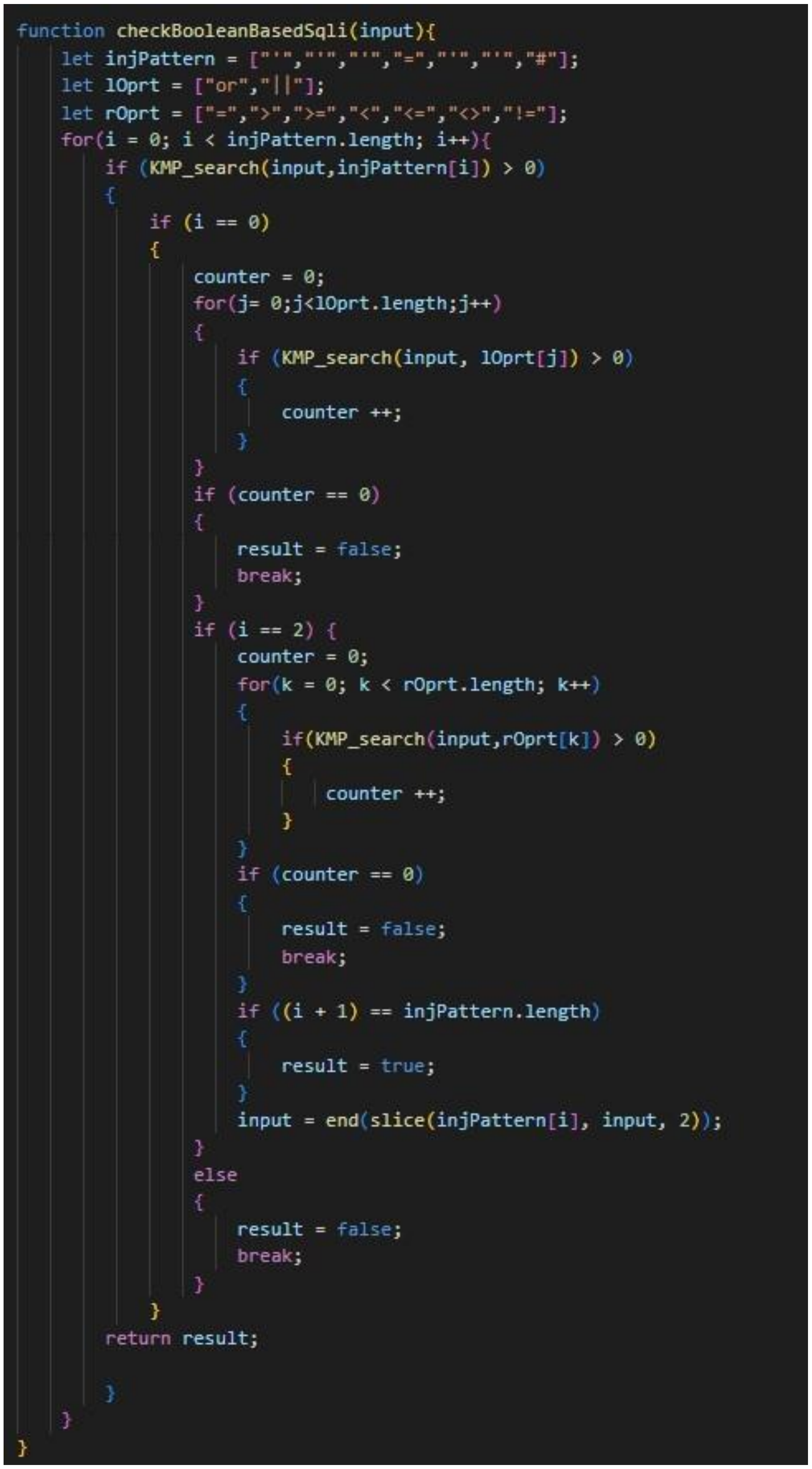
(v) Formulating the checkXss( ) function: this was used to prevent XSS attacks.

Implementation



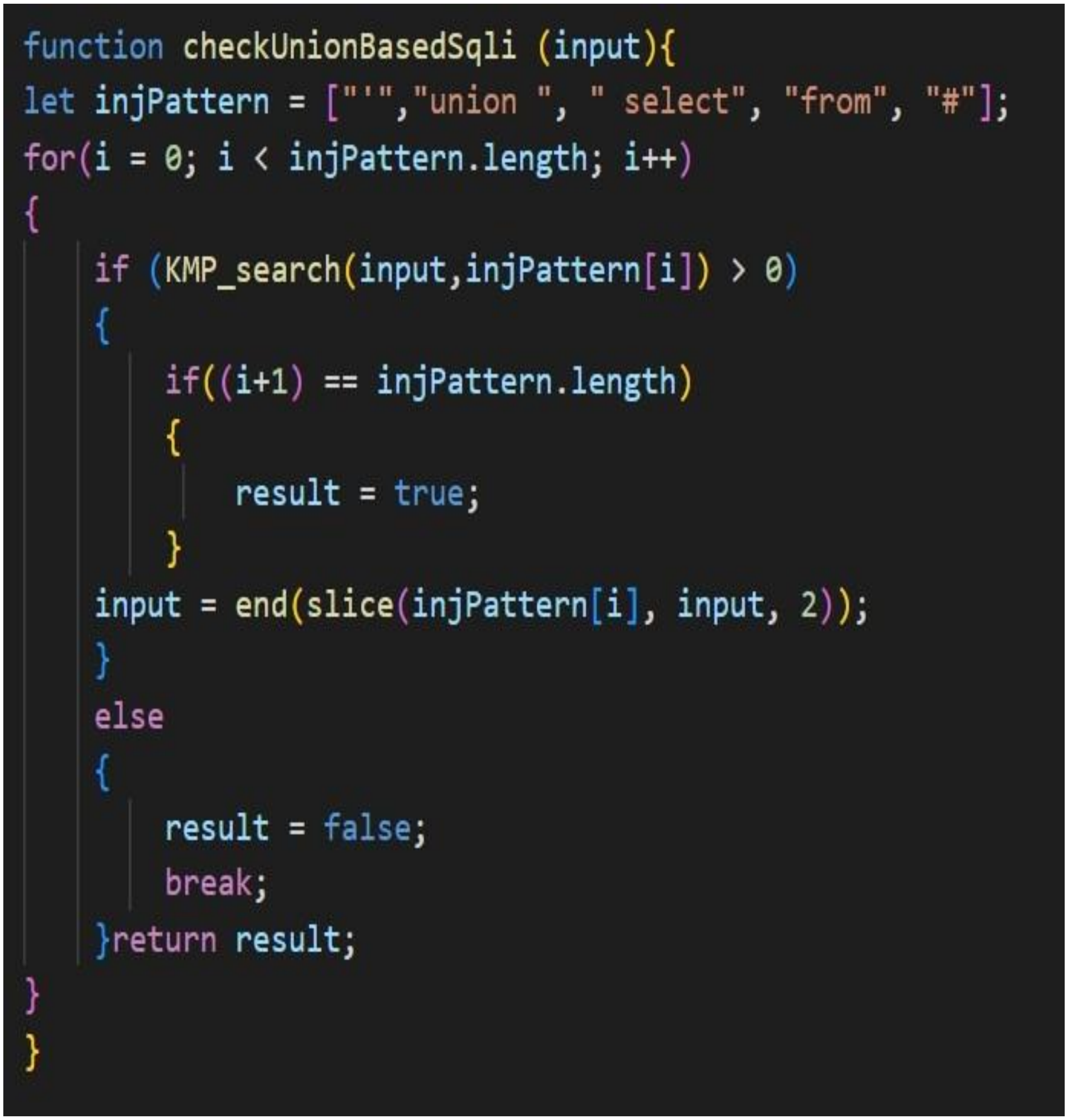
```
function filterForm()
{
    let l = document.forms["myForm"]["passw"].value;

     data = convertASCIItoString(l);
     if(data!="")
     {
         let a = checkBooleanBasedSqli(data);
         let b = checkUnionBasedSqli(data);
         let c = checkErrorBasedSqli(data);
         let d = checkBatchQuerySqli(data);
         let e = checkLikeBasedSqli(data);
         let f = checkXss(data);
         if(true (a||b||c||d||e||f))
         {
          alert("SQLI/XSS Attack Detected!");
             return false;
         }
         else
         {
             return true;
         }
     }
}
```

Figure 1: checking for attacks



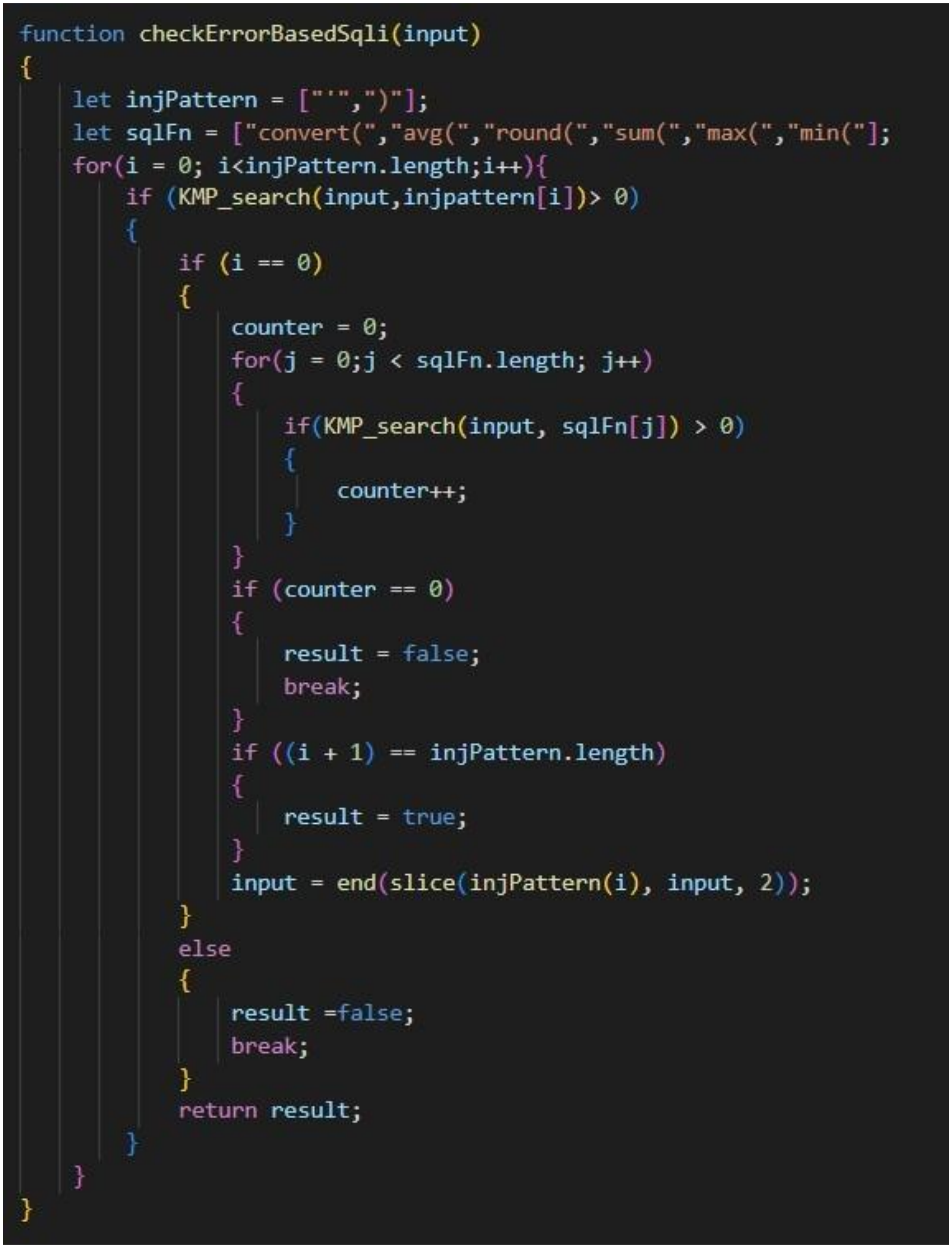
```
function checkBooleanBasedSqli(input){
    let injPattern = ["'","'","'","=","'","'","#"];
    let lOprt = ["or","||"];
    let rOprt = ["=",">",">=","<","<=","<>","!="];
    for(i = 0; i < injPattern.length; i++){
        if (KMP_search(input,injPattern[i]) > 0)
        {
            if (i == 0)
            {
                counter = 0;
                for(j= 0;j<lOprt.length;j++)
                {
                    if (KMP_search(input, lOprt[j]) > 0)
                    {
                        counter ++;
                    }
                }
                if (counter == 0)
                {
                    result = false;
                    break;
                }
                if (i == 2) {
                    counter = 0;
                    for(k = 0; k < rOprt.length; k++)
                    {
                        if(KMP_search(input,rOprt[k]) > 0)
                        {
                            counter ++;
                        }
                    }
                    if (counter == 0)
                    {
                        result = false;
                        break;
                    }
                    if ((i + 1) == injPattern.length)
                    {
                        result = true;
                    }
                    input = end(slice(injPattern[i], input, 2));
                }
                else
                {
                    result = false;
                    break;
                }
            }
        return result;

        }
    }
}
```

Figure 2: Checking for Boolean based SQLI attacks



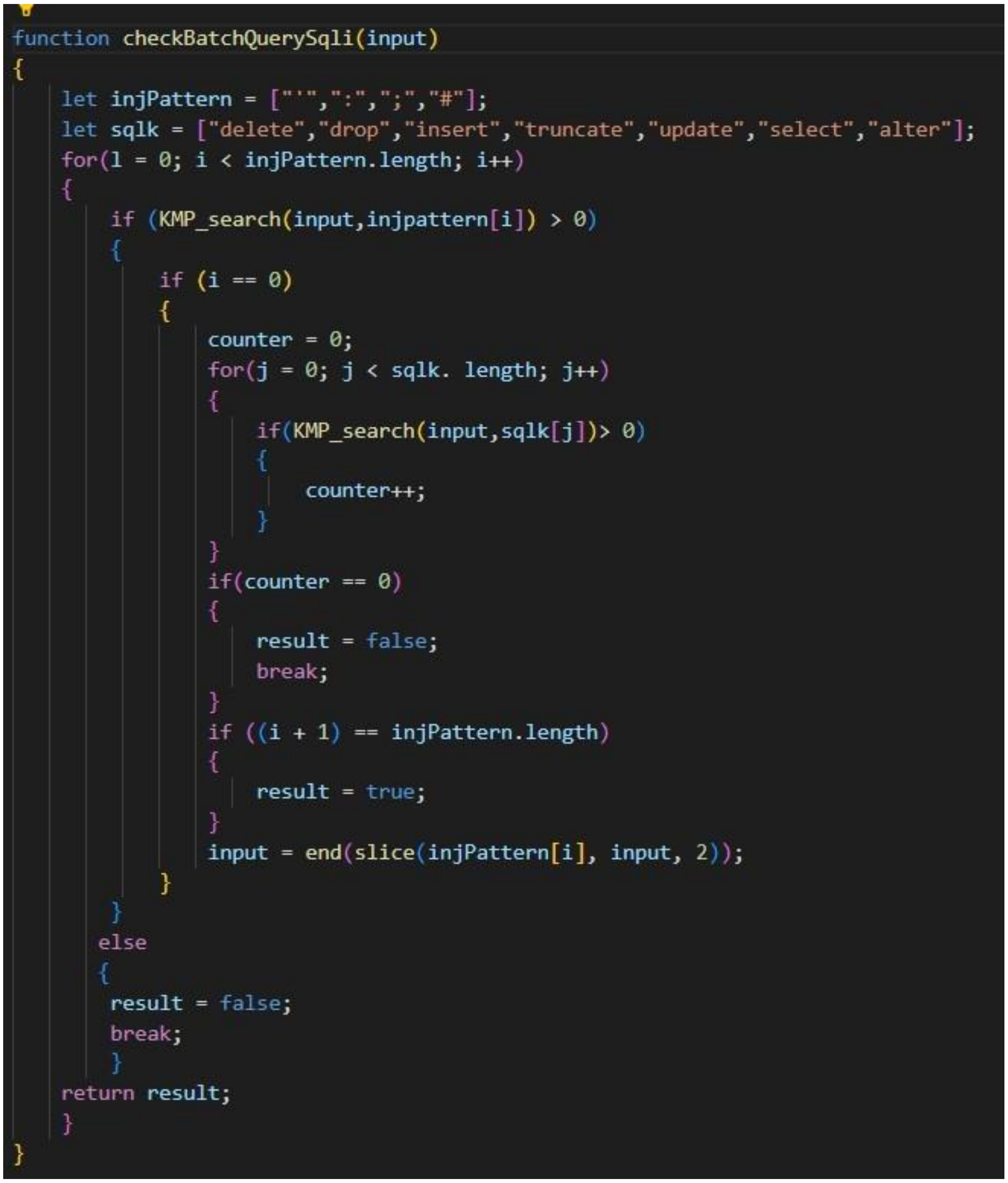
```
function checkUnionBasedSqli (input){
let injPattern = ["'","union ", " select", "from", "#"];
for(i = 0; i < injPattern.length; i++)
{
    if (KMP_search(input,injPattern[i]) > 0)
    {
        if((i+1) == injPattern.length)
        {
            result = true;
        }
    input = end(slice(injPattern[i], input, 2));
    }
    else
    {
        result = false;
        break;
    }return result;
}
}
```

Figure 3: Checking for union based SQLI attacks



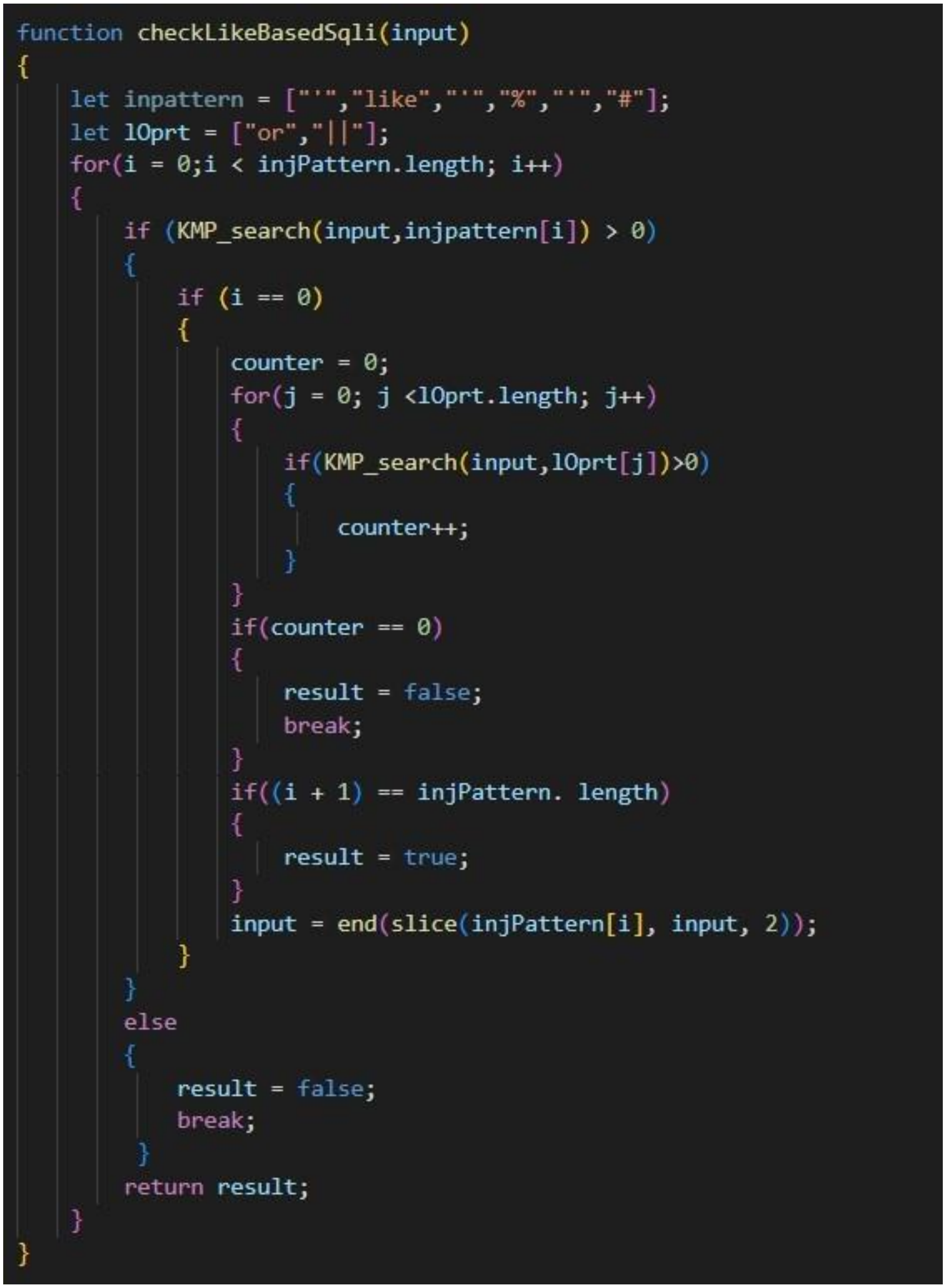
```
function checkErrorBasedSqli(input)
{
    let injPattern = ["'",")"];
    let sqlFn = ["convert(","avg(","round(","sum(","max(","min("];
    for(i = 0; i<injPattern.length;i++){
        if (KMP_search(input,injpattern[i])> 0)
        {
            if (i == 0)
            {
                counter = 0;
                for(j = 0;j < sqlFn.length; j++)
                {
                    if(KMP_search(input, sqlFn[j]) > 0)
                    {
                        counter++;
                    }
                }
                if (counter == 0)
                {
                    result = false;
                    break;
                }
                if ((i + 1) == injPattern.length)
                {
                    result = true;
                }
                input = end(slice(injPattern(i), input, 2));
            }
            else
            {
                result =false;
                break;
            }
            return result;
        }
    }
}
```

Figure 4: Checking for error based SQLI attacks



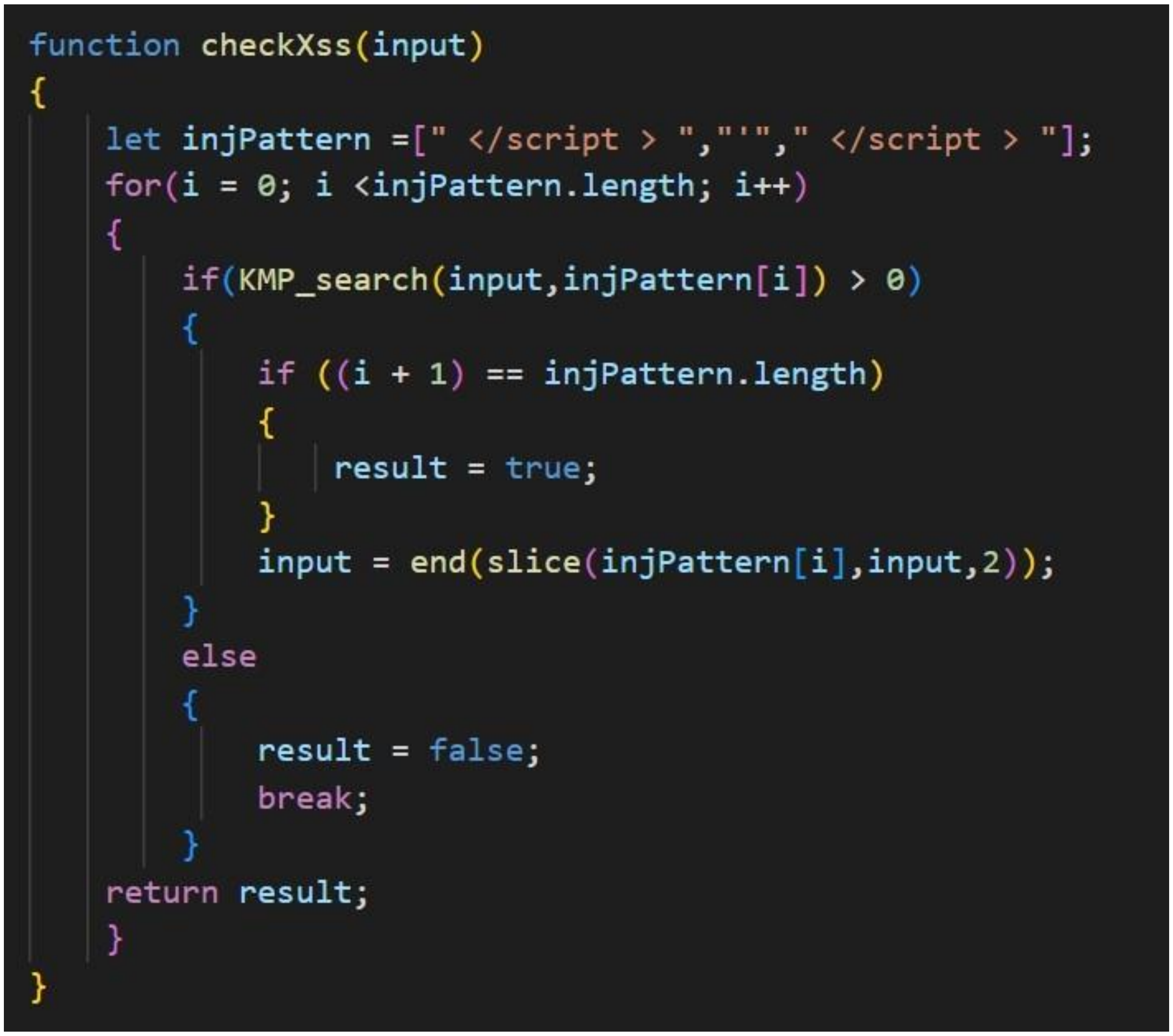
```
function checkBatchQuerySqli(input)
{
    let injPattern = ["'",":",";","#"];
    let sqlk = ["delete","drop","insert","truncate","update","select","alter"];
    for(l = 0; i < injPattern.length; i++)
    {
        if (KMP_search(input,injpattern[i]) > 0)
        {
            if (i == 0)
            {
                counter = 0;
                for(j = 0; j < sqlk. length; j++)
                {
                    if(KMP_search(input,sqlk[j])> 0)
                    {
                        counter++;
                    }
                }
                if(counter == 0)
                {
                    result = false;
                    break;
                }
                if ((i + 1) == injPattern.length)
                {
                    result = true;
                }
                input = end(slice(injPattern[i], input, 2));
            }
        }
       else
       {
        result = false;
        break;
        }
    return result;
    }
}
```

Figure 5: Checking for batch query based SQLI attacks



```
function checkLikeBasedSqli(input)
{
    let inpattern = ["'","like","'","%","'","#"];
    let lOprt = ["or","||"];
    for(i = 0;i < injPattern.length; i++)
    {
        if (KMP_search(input,injpattern[i]) > 0)
        {
            if (i == 0)
            {
                counter = 0;
                for(j = 0; j <lOprt.length; j++)
                {
                    if(KMP_search(input,lOprt[j])>0)
                    {
                        counter++;
                    }
                }
                if(counter == 0)
                {
                    result = false;
                    break;
                }
                if((i + 1) == injPattern. length)
                {
                    result = true;
                }
                input = end(slice(injPattern[i], input, 2));
            }
        }
        else
        {
            result = false;
            break;
         }
        return result;
    }
}
```

Figure 6: Checking for like based SQLI attacks



```
function checkXss(input)
{
    let injPattern =[" </script > ","'"," </script > "];
    for(i = 0; i <injPattern.length; i++)
    {
        if(KMP_search(input,injPattern[i]) > 0)
        {
            if ((i + 1) == injPattern.length)
            {
                result = true;
            }
            input = end(slice(injPattern[i],input,2));
        }
        else
        {
            result = false;
            break;
        }
    return result;
    }
}
```

Figure 7: Checking for XSS attacks

Results

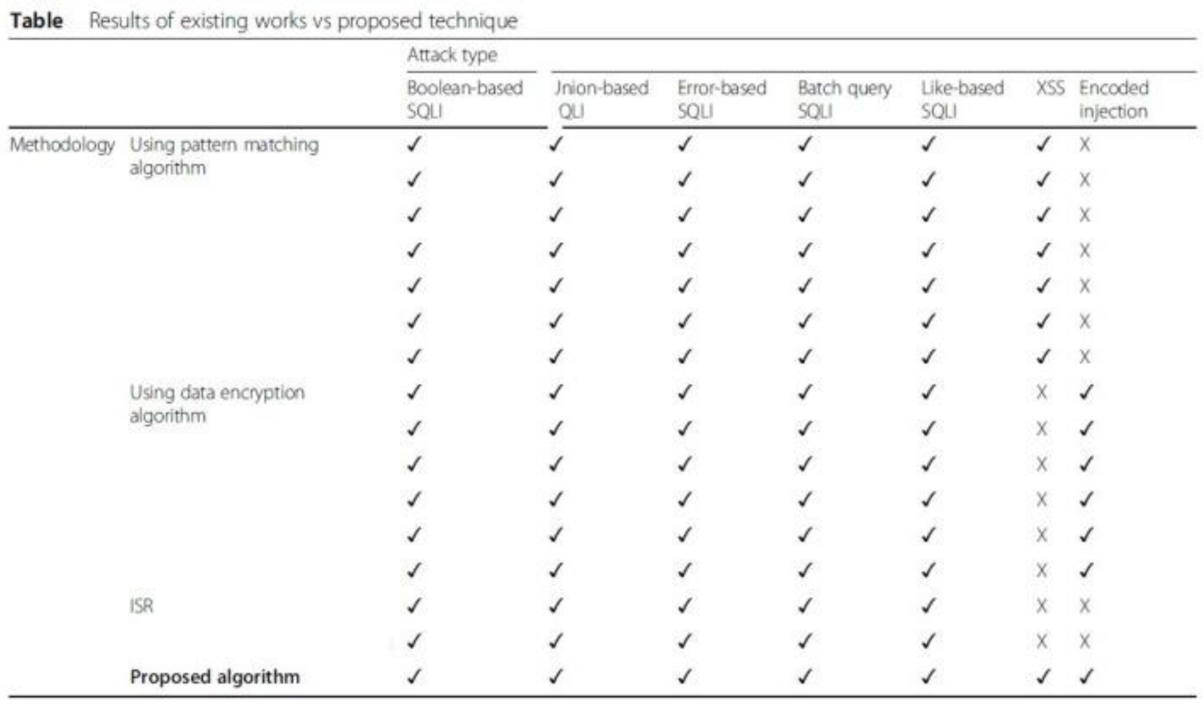


Table Results of existing works vs proposed technique

		Attack type						
		Boolean-based SQLI	Jnion-based QLI	Error-based SQLI	Batch query SQLI	Like-based SQLI	XSS	Encoded injection
Methodology	Using pattern matching algorithm	✓	✓	✓	✓	✓	✓	X
		✓	✓	✓	✓	✓	✓	X
		✓	✓	✓	✓	✓	✓	X
		✓	✓	✓	✓	✓	✓	X
		✓	✓	✓	✓	✓	✓	X
		✓	✓	✓	✓	✓	✓	X
		✓	✓	✓	✓	✓	✓	X
	Using data encryption algorithm	✓	✓	✓	✓	✓	X	✓
		✓	✓	✓	✓	✓	X	✓
		✓	✓	✓	✓	✓	X	✓
		✓	✓	✓	✓	✓	X	✓
		✓	✓	✓	✓	✓	X	✓
		✓	✓	✓	✓	✓	X	✓
	ISR	✓	✓	✓	✓	✓	X	X
		✓	✓	✓	✓	✓	X	X
	Proposed algorithm	✓	✓	✓	✓	✓	✓	✓

Table: Comparison of techniques

Results obtained from the proposed technique were compared with those available in existing literature. existing techniques which used data encryption algorithms were able to prevent all the five forms of SQL injection attacks but failed to prevent XSS attacks. Existing techniques which used PHP escaping functions and pattern matching algorithms were able to prevent all the five forms of SQL Injection attacks including the XSS attacks but failed to prevent encoded injection attacks. Existing techniques that used instruction set randomization were able to prevent all the five forms of SQL injection but failed to prevent XSS and encoded injection attacks. In a nut shell, any of the existing methods has its own drawback while the proposed algorithm prevents all the five forms of SQL injection attacks including XSS and encoded injection attacks.

Conclusion

A novel approach to detect and prevent SQL injection and XSS attacks is presented in this paper. The various types and patterns of the attacks were first studied, then a parse tree was designed to represent the patterns. Based on the identified patterns, a filter() function was formulated using the KMP string matching algorithm. The formulated filter() function detects and prevents any form of SQL injection and XSS attacks. Every input string is expected to pass through this filter () function. If at least one function returns True, then, the filter() function will block that user, reset the HTTP request, and display a corresponding warning message. The technique was tested using a test plan that consist of different forms of Boolean-based, union-based, error-based, batch query, like-based, encoded SQL injections and cross-site scripting attacks. The test results show that the technique can successfully detect and prevent the attacks, log the attack entry in the database, block the system using its Mac Address to prevent further attack, and issue a blocked message. A comparison of the proposed technique with existing techniques revealed that the proposed technique is more efficient because it is not limited to a particular form of attack, and it can handle different forms of SQL injection and XSS attacks.

References

[1] K. Elshazly, "A survey of SQL injection attack detection and prevention," *Journal of Computer and Communications,* 2014.

[2] N. Y. Xuan, "Securing e-commerce against SQL injection, cross site scripting and broken authentication," *Journal of Applied Technology and Innovation,* 2021.

[3] Meena, "Protection of E-Commerce Website from SQL Injection: A Review," *International Journal of Scientific & Engineering Research,* 2016.

[4] F. Q. Kareem, "SQL Injection Attacks Prevention System Technology: Review," *AJRCS,* 2021.

[5] I. Jemal, "SQL Injection Attack Detection and Prevention Techniques Using Machine Learning," *IJAER,* 2020.

[6] O. C. Abikoye, "A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm," *EURASIP Journal on Information Security,* 2020.

[7] J. P. Singh, "Analysis of SQL injection detection techniques," *CIISE,* 2016.

[8] D. Kar, "Detection of SQL injection attacks using Hidden Markov Model," *IEEE International Conference on Engineering and Technology,* 2016.

[9] R. Johari, "A survey on web application vulnerabilities (SQLIA, XSS) exploitation and security engine for SQL injection," *IJATER,* 2012.

[10] K. L.M., "SQL Injection Prevention System," *International Conference "Radio Electronics & InfoCommunications,* 2016.

[11] G. M, "Semantic Query-Featured Ensemble Learning Model for SQL-Injection Attack Detection in IoT-Ecosystems," *IEEE Transactions on Reliability,* 2022.

[12] S. Zivanic, "Network security education: SQL injection attacks," *IEEE Zooming Innovation in Consumer Technologies Conference*, 2022.

[13] S. Kini, "SQL Injection Detection and Prevention using Aho-Corasick Pattern Matching Algorithm," *3rd International Conference for Emerging Technology*, 2022.

[14] A. M. A. M. Samar M. Albalawi, "Authentication Enhancement Against SQL Injection Attacks (SQLIAs)," *2nd International Conference on Computing and Information Technology*, 2022.

[15] S. S. S. K. V. G. N. M.S. Roobini, "Detection of SQL Injection Attack Using Adaptive Deep Forest," *International Conference on Communication, Computing and Internet of Things*, 2022.

[16] M. O. A. G. I. A. M. D. M. Marion Olubunmi Adebiyi, "An Sql Injection Detection Model Using Chi-Square with Classification Techniques," *International Conference on Electrical, Computer and Energy Technologies*, 2021.

[17] C. H. W. X. J. L. Pengcheng Wen, "SQL Injection Detection Technology Based on BiLSTM-ATTENTION," *4th International Conference on Robotics, Control and Automation Engineering ,* 2021.

[18] D. Lu, "A GAN-based Method for Generating SQL Injection Attack Samples," *IEEE ITAIC,* 2022.

[19] L. Min, "The Detection and Defense Mechanism for SQL Injection Attack Based on Web Application," *Joint International Information Technology and Artificial Intelligence Conference ,* 2022.

[20] P. Roy, "SQL Injection Attack Detection by Machine Learning Classifier," *International Conference on Applied Artificial Intelligence and Computing*, 2022.